FLASH Performance Experiment Notes

RTFlame problem

The location of a production run (containing parameter files + checkpoint files) is specified below. This uses 6 levels of refinement in a 3D AMR grid. A simulation of this resolution is run on 4096 nodes (16384 processors (VN mode)) on BG/P.

Setup line: ./setup RTFlame -3d -auto -nxb=16 -nyb=16 -nzb=16 +parallelio -unit=Particles/ParticlesMain?

Points from discussion with RTFlame simulation developer:

We are only interested in certain fixed problem sizes. These are specified by the resolution:

```
256 resolution - lrefine_max = 5,
512 resolution - lrefine_max = 6,
1024 resolution - lrefine_max = 7
```

(Weak scaling does not make sense for RTFlame).

Email describing location of checkpoint file of interest

```
You mentioned you'd like a pointer for something of the proper scale
for the PERI work.  So here you go...

On BG/P a good place to look for an example run is:
/gpfs1/townsley/runs/RTFlame_512_s6/Run53

This is at just after 0.28 seconds, so it is very well-developed.
(note that the restart file here is a symbolic link into ../Run52).
This should give you a good example flash.par, a restart file and
other stuff to use.  This leg was done on 16k processors.  As
mentioned in the meeting, a bit different than I had said to you in
person, it probably makes the most sense to have the PERI work
target 32k processors for this problem.  At that level I think we
would be taking a pretty serious hit due to scaling (i.e. > 30%,
possibly significantly more), and if that could be ameliorated it
we would use it.  This is true for any possible future RTFlame runs
but more importantly for the stirred runs which will have a similar
mesh geometry.

Again, for the purposes of this discussion we are assuming IO is a
separate issue.
```

There are approximately 10 blocks per processor at 16,384 processors in the 512 resolution problem (above). At this scale, the developer believes there is a load imbalance issue which affects performance, e.g. some processors have 8 blocks per processor and others have 10 blocks per processor. However, he mentions the real limiting factor in these simulations is IO. Note, the IO is neccessary because we must track the evolution of the simulation. (He has not tried the split IO mode in FLASH, in which we still use parallelIO, but write to multiple files).

The developer uses: 2048 processors for 256 resolution, 16,384 processors for 512 resolution, and has not yet run 1024 resolution.

(A restart file for 512 resolution is approximately 70GB).

WD_def problem

We generated a weak scaling curve (WD_weakscaling.pdf) on BG/P using data from 4416, 6144, 8192 processor runs:

```
Updated numbers, all tests run with updated code, with Paramesh4dev:

Nblocks(ini) Nprocs t_evo  tref         max#blk/proc mem/proc[GiB]
-------------------------------------------------------------
110387       4416   573.8  32.5         29              .5
110387       8192   374.1  24.8         17              .5
154419       6144   609.8  47.5         29              .5
203443       8192   633.9  55.2         29              .5


t_evo is the evolution time for 10 steps.  This is not the
total time, i.e. it does not include initialization.
Rows 1,3,4 constitute the weak scaling curve to be.
Rows 1,2 test strong scaling.



Setup syntax: ./setup WD_def -3d +cube16 -maxblocks=70 +noio -auto +pm4dev -objdir=PM4dev -parfile=[spe
Here, "specific flash.par" is:
scaling5_intrepid.par - Row 1 (4416 procs) & Row 2 (8192 procs) in table.
scaling5h_intrepid.par - Row 3 (6144 procs) in table.
scaling6_intrepid.par - Row 4 (8192 procs) in table.
```

For the weak scaling we attempt to keep the number of blocks per processor approximately constant. In simpler problems, it is possible to just change the lrefine_max in flash.par to get to the next problem size. Then keep adjusting the number of processors until you get roughly the same number of blocks per processor. However, with WD_Def the amount of refinement is hard to calculate when varying lrefine_max. Therefore, we vary r_match & uni_radius in the flash.pars to adjust the problem size. We do not vary the refinement maximum level. It required quite a bit of work to come up with these values. They were chosen to give problem sizes of (roughly) 100 * 64 * 2**(n-1) total blocks, for integer n, n is the number in scaling<n>.

Note, weak scaling should be easier on the XT4 since there is lots more memory per processor. Also, eliminate IO from the runs, that should give you more memory to play around in.

More details, which I include in their raw format:

```
First, an updated summary of WD_def scaling runs with the
same code as used previously (2007) on Seaborg and Franklin.
This is the same table I sent earlier, except for
 - some reordering,
 - elimination of some failed runs,
 - addition of the test with 154419 blocks .
Note that that mem/proc column shows whether a test was done
in virtual mode (0.5 GiB/proc) or dual mode (1.0 GiB/proc).

Nblocks(ini) Nprocs t_evo tref t/1mort max#blk/proc mem/proc[GiB]
=============================================================
```

```
  6067        256   567.9  47.9   0.45  29*             .5       *27 up to last step
  6067        240   598.6  51.6   0.43  30*             .5       *28 up to last step

 12083        480   636.0  57.2   1.87  29              .5

 54515       2176   613.0  66.7   4.36  29              .5

110387       3800                      32             1.0       **failed
110387       4096                                      .5       **failed
110387       4096   647.1  61.9   8.61  30             1.0
110387       4416   609.8  59.5   8.88  29              .5
110387       8192   396.9  42.2  11.25  17              .5

154419       6144   663.6  91.2  12.31  29              .5

203443       8136   700.9 106.7  16.86  28(leaf:24)     .5
203443       8192   699.9 107.7  17.25  29(leaf:23)     .5

Times are from FLASH timers, given in seconds:
t_evo  = total evolution time for 10 steps
tref   = total evolution time spent in Grid_updateRefinement
t/1mort= time spent per invocation of amr_morton_process

As I noted before, these tests show relatively poor scaling
when going from ~4k to ~8k procs.  (The new ~6k test doesn't
add anything new, it fits the trend.)  Again summarizing
what I could figure out from looking at the FLASH timers:
The increase in t_evo is due mostly to increased time spent
in Grid_updateRefinement and, to a lesser degree, increased
time spent in Hydro and sourceTerms (and within those, the
time increase is mostly due to time spent in amr_guardcell
calls).

Increased time required in Grid_updateRefinement is something
we saw before on Franklin, in runs with ~800,000 blocks on
~8k procs and larger.  So on both machines, intrepid and
franklin, we have evidence of degraded scaling in
Grid_updateRefinement with an onset at ~8k procs.
It used to be assumed that on franklin, this was due to
that architecture's intrinsically poor scaling of global
operations like MPI_AllToAll and MPI_AllReduce.  So it
was unexpected to find the same kind of behavior on BGP.

The timer info from franklin and now BGP also shows that
the part of Grid_updateRefinement where scaling breaks down
is within the PARAMESH routine amr_refine_derefine,
and within that in the routine amr_morton_process.
And if it's not architecture-specific quirks in the
behavior of global MPI calls that degrade performance here,
then the algorithms used here by PARAMESH may just not
be scaling well.

Kevin Olson has basically rewritten the offending parts of
PARAMESH code for PARAMESH 4.1.  We have had the trunk FLASH
code working with that version of PARAMESH (called Paramesh4dev
within the source tree) for a while, but haven't systematically
tested performance.  I decided to test whether weak scaling in
WD_def was improved by using this newer code.  And of course,
the newer code should not only scale better, but also be at
least as efficient as the previously tested version.
```

I made the necessary changes in several steps. The following is
mostly narrative, but I'll show total evolution and refinement times,
same measure t_evo, and tref as above, for 203443 blks on 8192 procs
after each step, and also for 110387 blks on 4416 procs (or smaller)
where available.

1. Updated the code for testing to current trunk level.
A (surprisingly painless) Subversion merge.

Nblocks(ini)	Nprocs	t_evo	tref	t/1mort	max#blk/proc	mem/proc[GiB]
54515	2176	760.6	44.1	4.36	29	.5
203443	8192	834.7	83.7	17.27	29	.5

It can be seen that the code has become significantly slower. However,
this is not really a surprise. We already knew that the trunk code
had become more inefficient (especially for WD?) at some point.

2. Removed unnecessary EOS calls from Hydro code.
Examination of timer info showed that the slowdown was
probably due to a change in the way EOS is called on guard
cells before each Hydro sweep. After reverting the logic
back to a previous code version:

110387	4416	567.7	35.8	8.85	29	.5
203443	8192	650.5	83.4	17.23	29	.5

It can be seen that the test now actually runs faster than the
originally tested version. (Not sure which changes exactly
are responsible for this improvement. Prabably careful removal
of unnecessary EOS calls in varius places play a large role.)

3. Compiled FLASH with "Paramesh4dev" instead of "Paramesh3".
(The trunk code is Paramesh4dev-ready, so adding the argument
+pm4dev to the setup command line is all it takes.) Results:

110387	4416	573.8	32.5	?	29	.5
203443	8192	633.9	55.2	?	29	.5

(The Paramesh4dev code was not instrumented with additional
timers, thus the time per call to PM4 amr_morton_process is
unknown.)
Note that Grid_updateRefinement actually changes the grid only
1 time in the 110387-block simulation, but 3 times in the 203443-block
simulation. This may account for the remaining difference in tref.

Conclusions:
A. The newer FLASH code (with the mentioned modification to Hydro)
is more efficient overall than the previously tested code; thus
presumale more efficient than the code derived from the wd_def
repository branch used in WD_def simulations up to now.

B. The newer FLASH code scales about the same as before when
using Paramesh3: (weak scaling, comparing time increase
for 110387-block -> 203443-block tests)

    699.9 / 609.8 = 1.15   (before)
    650.5 / 567.7 = 1.15   (new)

The newer FLASH code scales better when using Paramesh4dev:

    633.9 / 573.8 = 1.10

```
This is only a moderate improvement; the scaling may look
even better when comparing tests with the same number of
grid-change events. (TO DO)

C. I suggest to make the newer code with Paramesh4dev
the base of the scaling tests to be shown.  I will submit
runs to fill out the series.

D. The newer code - basically trunk - should become
the version used in WD simulations.

E. I suggest to make the newer code with Paramesh3 the
base of estimates for CPU requirements for future WD
runs on BGP.  This suggestion is based on the assumption
that WD runs on BGP will likely be run on up to ~4k procs
but not significantly more.
In particular, use 567.7 s / 10 = 56.7 s per time step
for "grind time" estimate. If desired, this number can easily be adjusted upward
for the fact that in real simulations, the grid usually
changes 5 times per 10 steps (rather than 1 time as
in the 110387-block test).
```

We have been running some wd_def simulations on BG/P and are having to make some real sacrifices to get it to work in VN mode. We wanted to run a simulation with 10,000 blocks. The largest number of blocks we could fit was about 20 per processor. At this size, the simulation was taking approximately 1 minute per timestep. In comparison, a 10,000 block simulation on Franklin with approximately 60 blocks per processor takes 40 seconds per timestep. We ran the same 10,000 block simulation on 2048 processors giving 3-6 blocks per processor and this takes 20 seconds per timestep. Memory is tight as the Multipole solver consumes about 70MB of memory in this simulation, and the addition of particles costs more memory still. CONCLUSION: To run simulations on BG/P (VN mode) we have to aim at less than 10 blocks/processor to fit in 512MB memory. At such a low count we expect the load balance issue to be more important as a single block counts for a noticable amount of total work on a processor.